Assignment 3 - Group 43

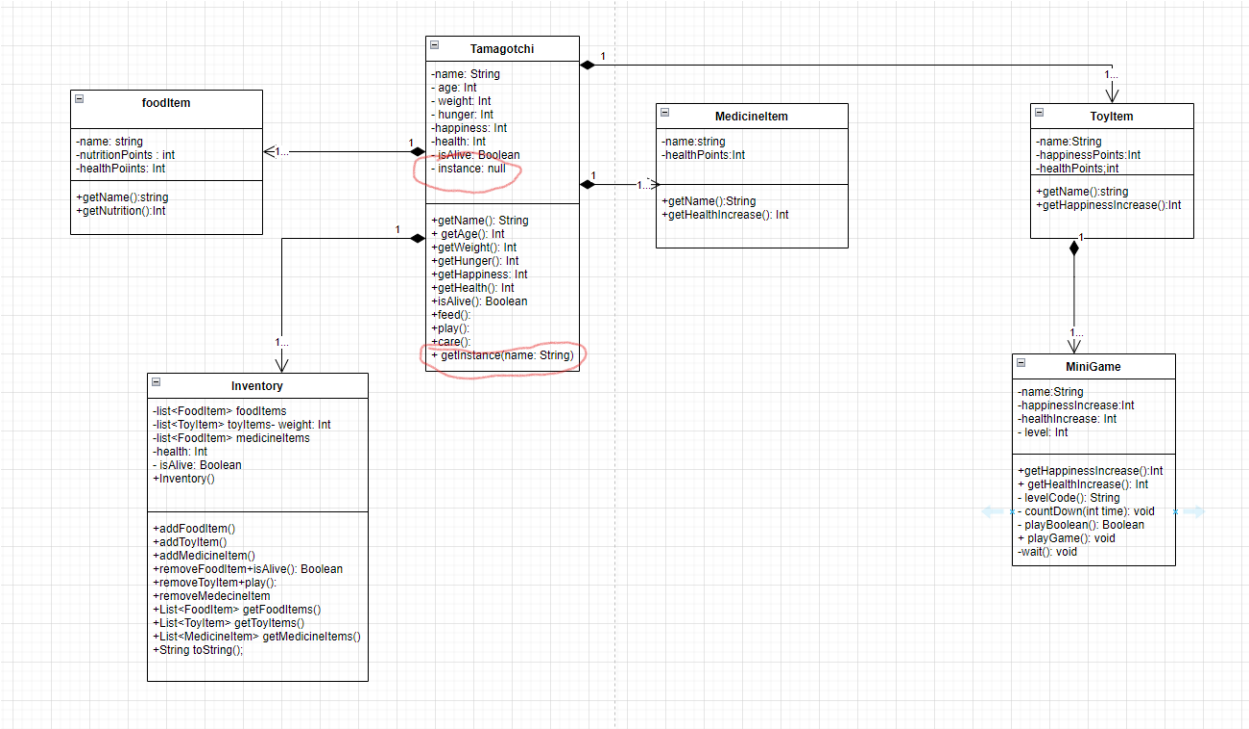
Name	Student number	Email
Sanam Izadi	2729395	s.izadi@student.vu.nl
Tanav Rawal	2725239	t.rawal@student.vu.nl
Filip Bickelhaupt	2713594	f.m.bickelhaupt@vu.nl
Denay Araya	2717149	d.araya@student.vu.nl

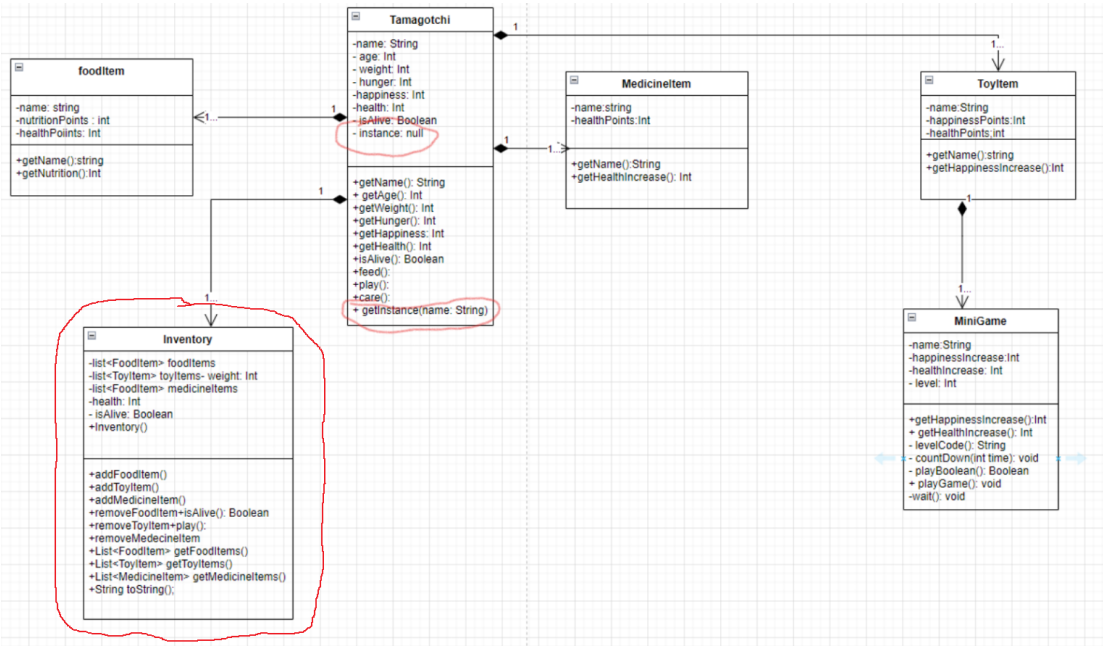
Summary of changes from Assignment 2

Author(s) - Tanav Rawal

- In regards to the class diagram we simplified the tamagotchi class so that it is easier to understand, and not a very deep class with all of the functions which were in assignment 2
- We also implemented a minigames class which allows for all the functionalities of a minigame in our tamagotchi game.
- There were not changes required with the object diagram.
- Put more guards and carefully thought of the events in the state diagram, and made the state machine diagram more refined.
- Added more features from sequence diagram to make the food system in the game applicable to medicine as well.

Revised class diagram





While coding we deviated quite a bit from the original plan since we didn't find everything as practical, useful, efficient or worthwhile as we thought while creating the diagrams. For the object diagram we changed a lot of the

Tamagotchi class but not alot of the other classes. In the tamagotchi class we got rid of the hygiene variable and the bathing method. We also got rid of the different types of tamagotchi by removing the “type” variable because we didn't know what different types of tamagotchis would entail and it over complicate the entire program. This means there's one type now. The tamagotchi also doesn't sleep anymore since there's no isAsleep() method anymore. We also removed the LastFedTime and lastPlayedTime because their use was not practical anymore, as we had a functioning timer which would show all the stats of the tamagotchi so the player would know exactly how their pet is doing at each stage of the game. From our previous assignment we had planned to use an inventory function which would store the different food items and toy items and medicine items, however, we understood that using this kind of method would be quite inefficient and would lead to a lot of redundant code, which would also make the whole code quite error prone and complicated. So we decided to change it to use a separate inventory class which basically stores the different items of food, medicine and toys or minigames that we could use. Apart from the different inventory class we have also made changes with the minigame class, we found that it is easier to implement a different class for the minigame, which is just connected to the toyItem class and inheriting from it, as we have a vastly different minigame which was making the code very complicated and long, so making a different class and adding the minigame as a basic option for the toys was a better option overall.

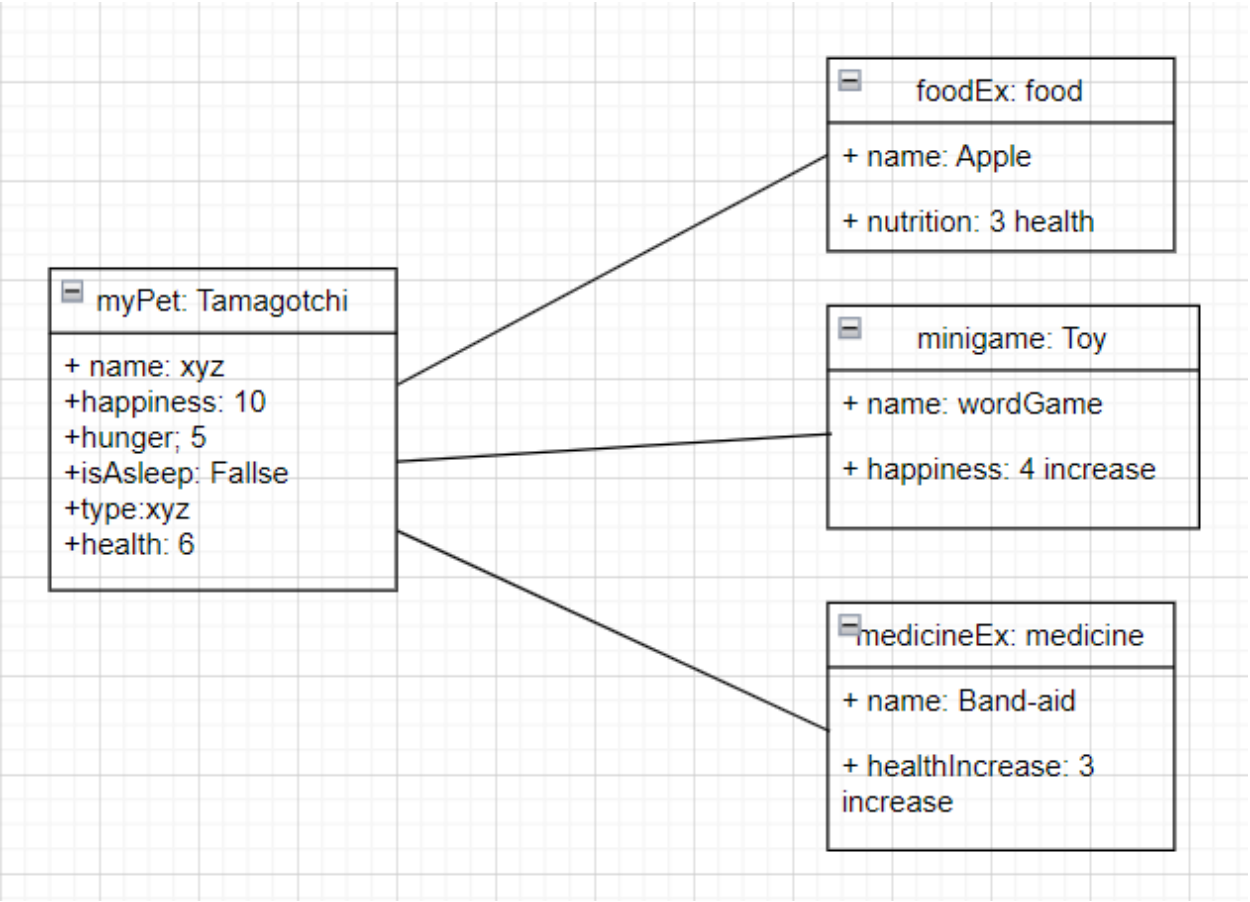
In terms of the application of the design we can see that from the diagram , the red highlighted part is the main implementation of the singleton design pattern, we created instance, which is private, and a public constructor so that will only take the same cached object that we have created in the first place rather than creating a new one in every instance of the game.

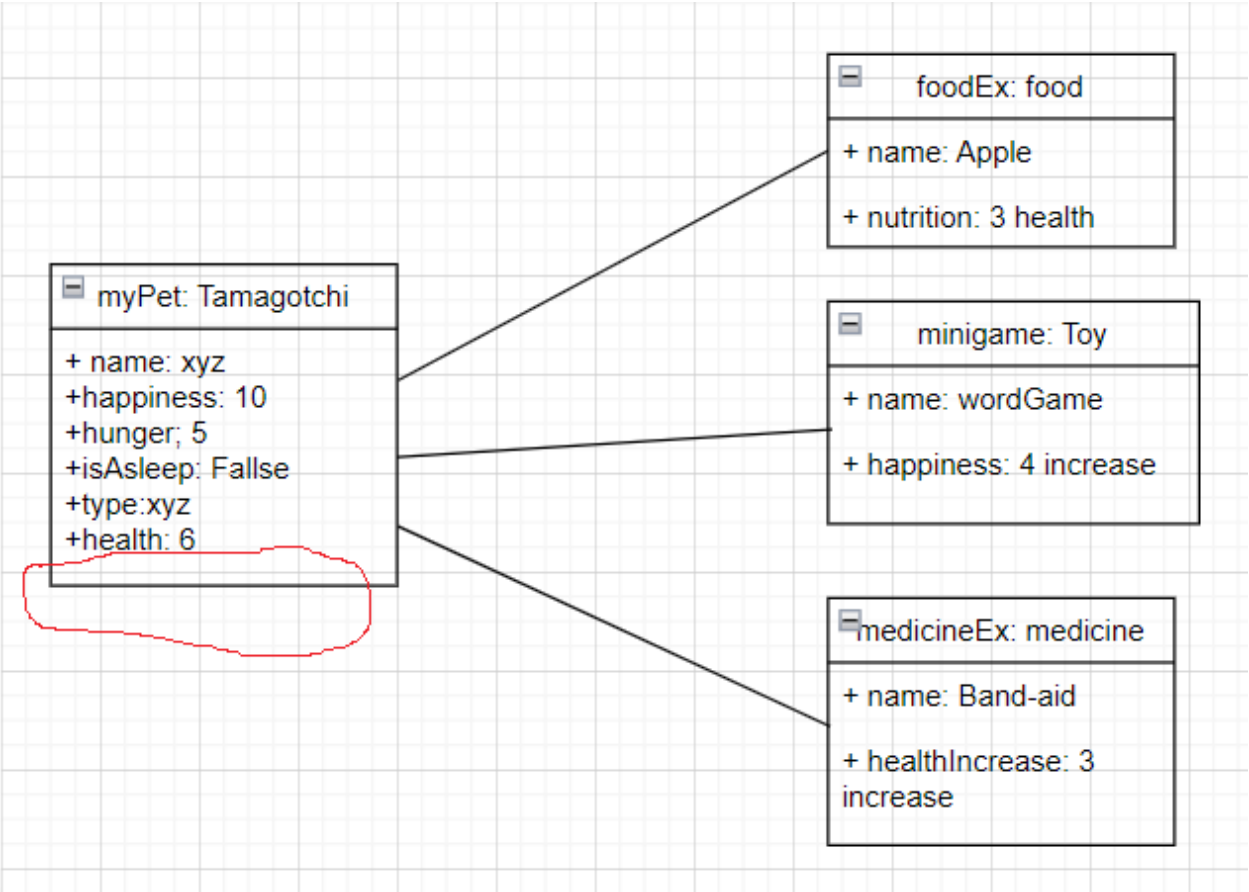
Application of the singleton design pattern

Author: Tanav Rawal, Sanam Izadi

	DP1
Design pattern	Singleton design pattern is applied
Problem	The points of access are controlled through the many object we create of the tamagotchi class. If we only make one point of access it will make our code and overall program flow very understandable and simple.
Solution	We need a couple steps to implement this design pattern 1) We first made the default constructor private, and prevent the other objects from using the new operator 2) We also create a static creation method, which will act as constructor for all of the objects which are being created.
Intended use	During run time we are creating a object of the tamagotchi class, which will be better in terms of the overall efficiency of the program, so instead of using the “new” operator we use the in built getInstance() function which will create one method, and only that method will be used during the whole program.
Constraints	No real constraints are needed with this application of the design pattern.
Additional remarks	Optional, only if needed

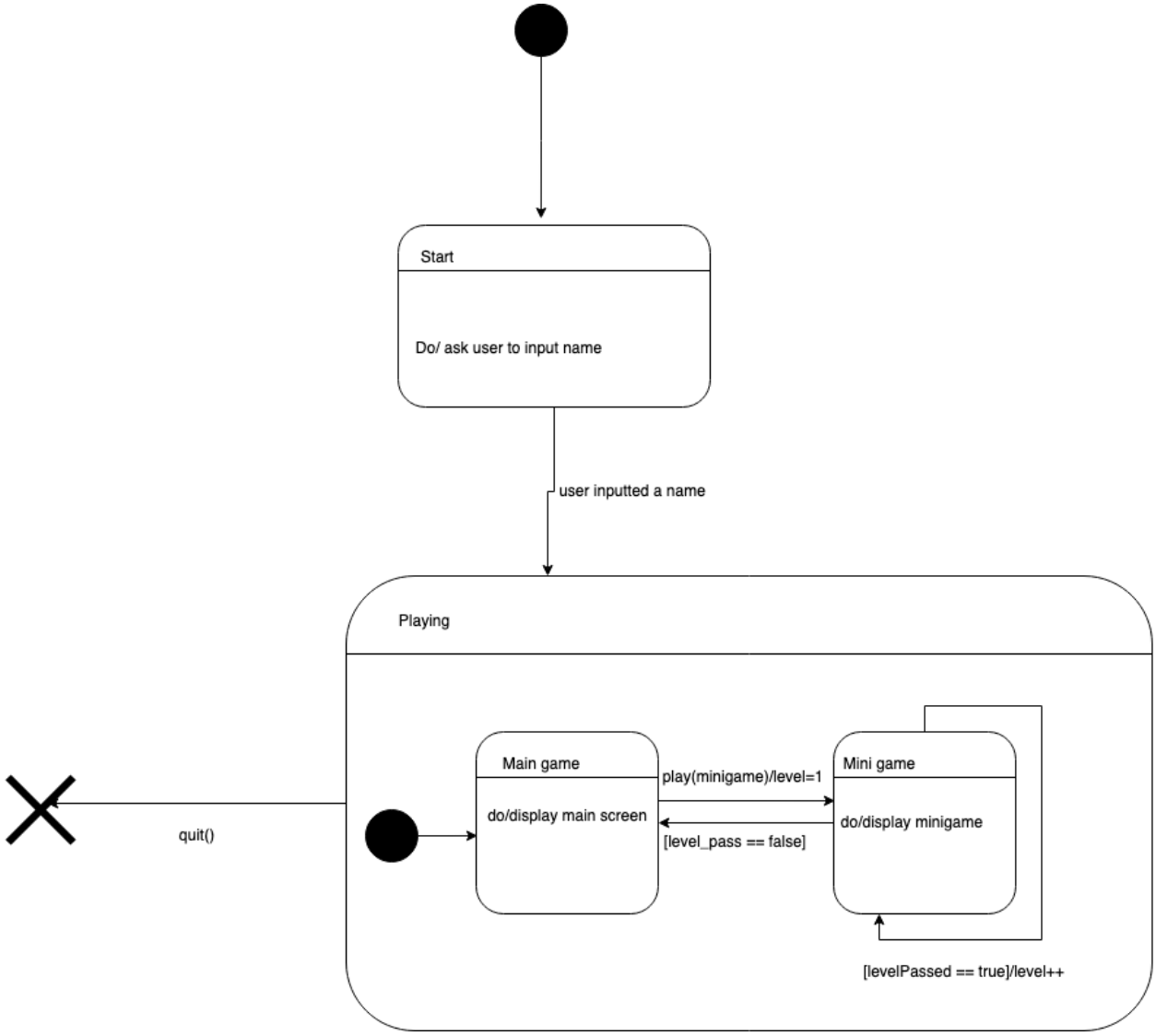
Revised object diagram





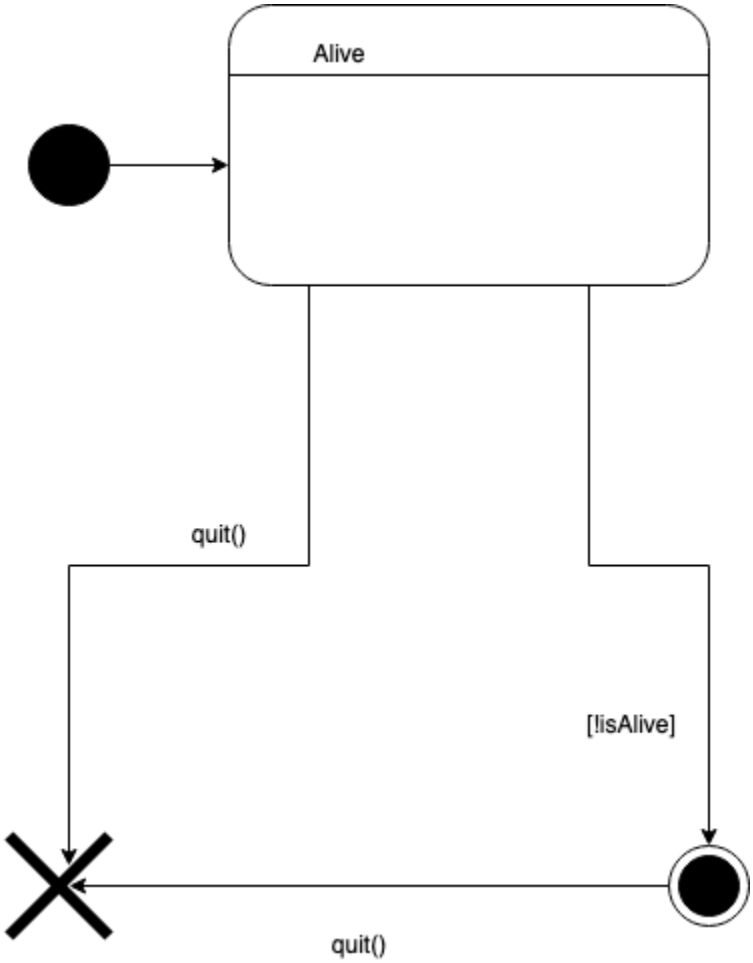
For the object diagram we changed very little. The only thing we changed is that we removed two methods from the myPet object from the Tamagotchi class. We removed the “+lastPlayedTime:” method and the “+lastFedTime:” method. We removed these because we already have a general timer for the entire program that keeps up the the past time since any given relevant event for the tamagotchi. So the two removed methods are redundant. The red circle in the image is where the two methods used to be. From the previous assignment the overall flow of the program is similar and all of the classes of medicine, toy and food are deeply connected to the tamagotchi class.

Revised state machine diagram 1



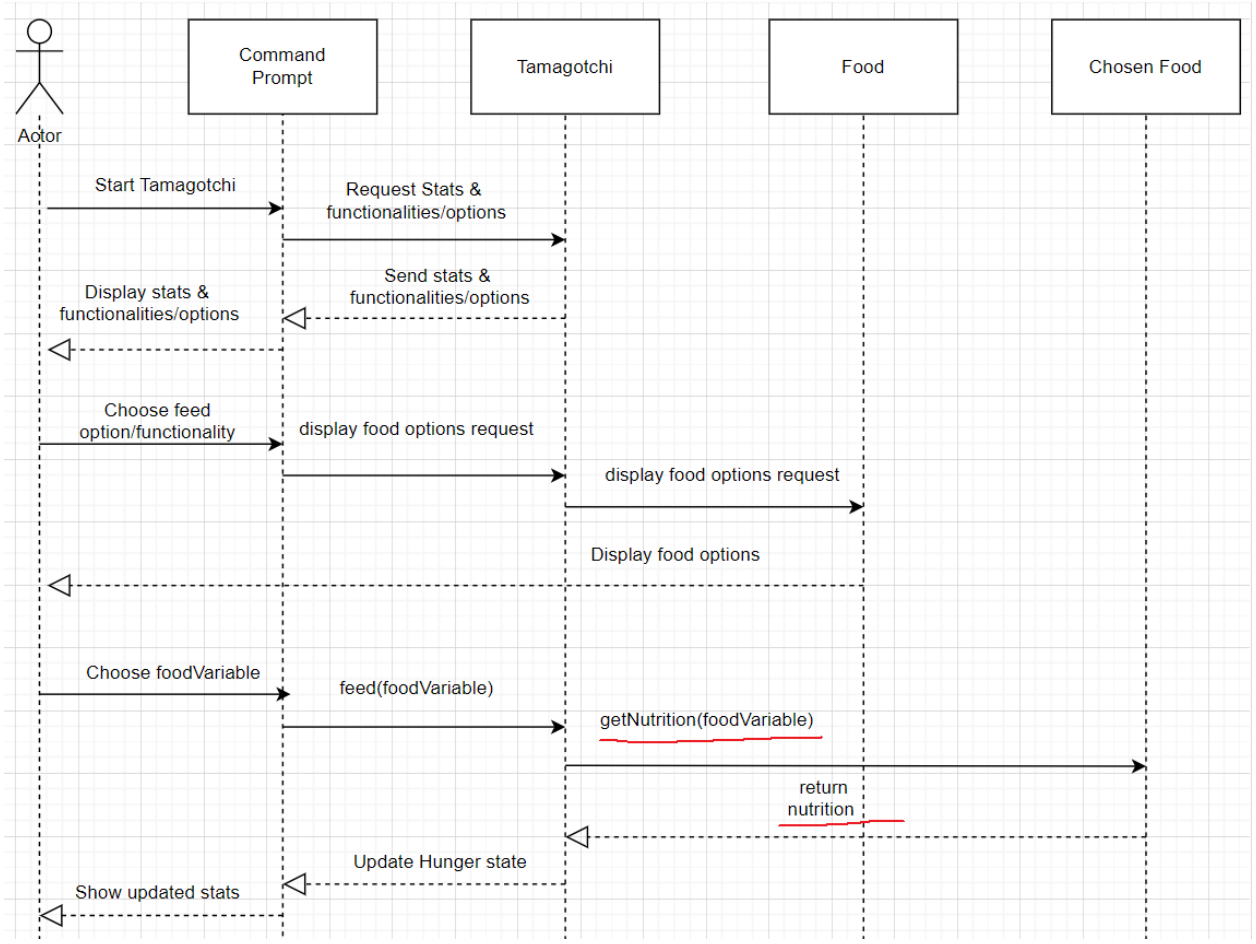
This is the revised diagram of the state of the tamagotchi. We decided to not implement the ability to have multiple tamagotchi pets at once and the ability to save your tamagotchi, when the program is closed the tamagotchi is lost. So now at the start screen the user is asked to input a name for their tamagotchi and the game will start. Inside the playing state is the same as before, but now the playing state is left by quitting the program.

Revised state machine diagram 2



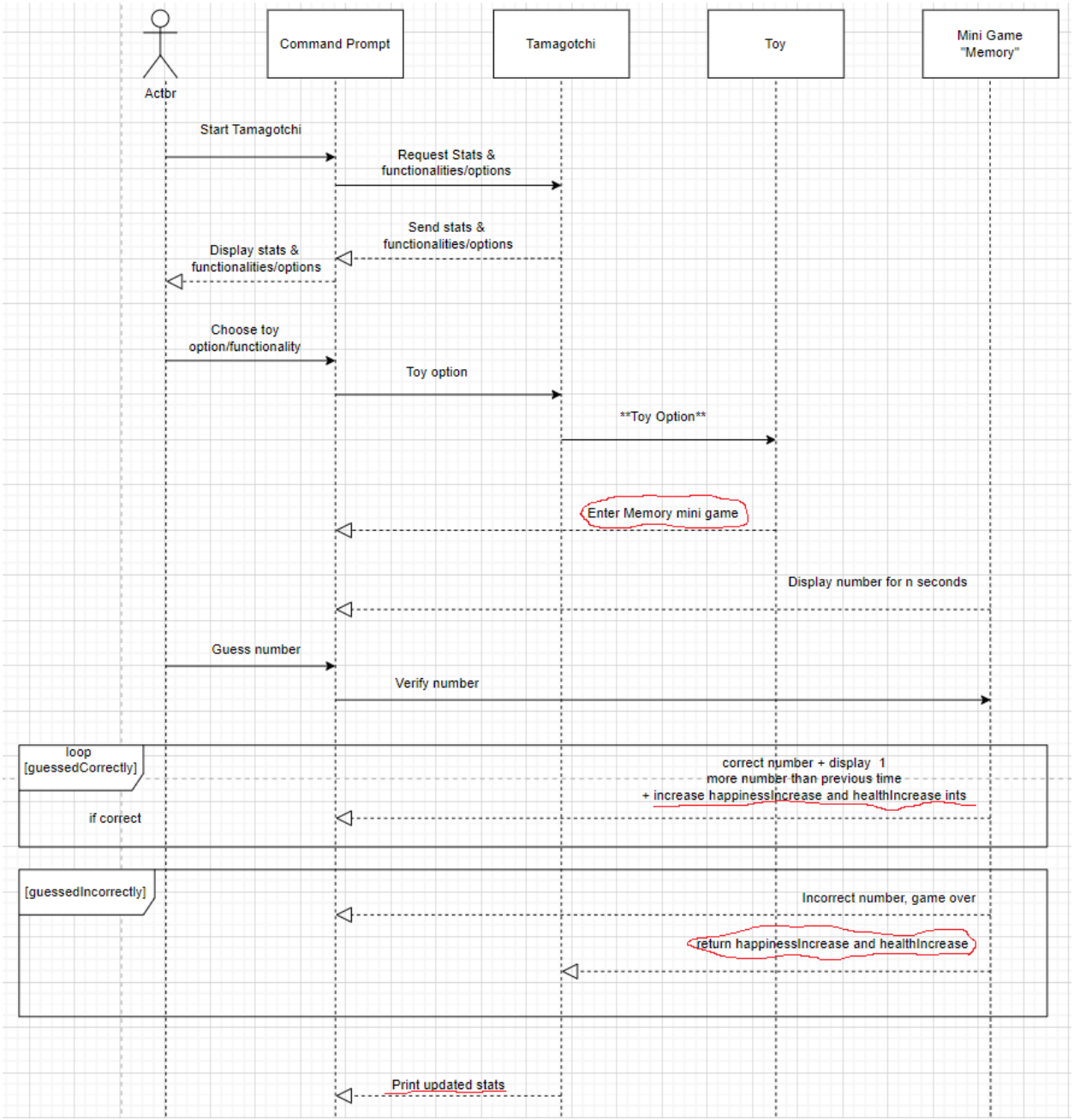
This is the revised diagram of the state of the game. We decided not to implement that the tamagotchi starts as an egg or the age states within the alive state. We also did not implement a delete function but instead the tamagotchi is deleted by quitting the program.

Revised sequence diagram 1 (feeding)



For the sequence diagram for feeding the tamagotchi we changed very little. All we really changed was the communication between the tamagotchi class and the chosen food object. Before, the tamagotchi class would ask the nutritional information from the food class, which would then ask this information to the chosen food object. The chosen food object would then respond by sending the requested information to the food class which would then pass it on to the tamagotchi class so that the hunger stat could be updated. The food class basically functioned as a middle man between the tamagotchi class and the chosen food object. However while programming we realized that the tamagotchi class directly communicates with the object and does not get redirected by the food class. So now you can see in the diagram that the two underlined commands at bottom right of the image go directly from tamagotchi to chosen food instead of stopping at the food class.

Revised sequence diagram 2 (minigame)



We changed a couple of things for the sequence diagram of the minigame. In the old diagram once the actor chooses the toy option, they would be able to choose the minigame they desired. But since we only have one minigame we thought this was not necessary. In the second underline command we added the line “increase happinesIncrease and healthIncrease ints.” Which means that when the actor fills in the correct input these two variables will increase in value. We added this since we need to keep track of how much health and happiness is gained from playing the game. In the third underlined command these same two variables get returned to the tamagotchi class once the actor fails the game. This is done so that the happiness and health stats in the tamagotchi class can be updated. Afterwards in the last underlined command

the updated stats get printed so that the user can see the affect the game had on the tamagotchi.

Implementation

Author(s): Tanav, Denay, Sanam, Filip

- **Strategy for implementation** - The main strategy that we used in order to convert from the UML models to working code in java, we first understood about the concepts that we would use in java to implement all the classes and how to implement the methods to accurately implement the different actions the tamagotchi can perform while the user plays. After coming to a common understanding about the methods and patterns we would use in java, we actually started to implement the tamagotchi class first, as every other class was connected to this one class, as it was basically the main pet and the main point of contact for the pet and the user. From this one main class we would control the behaviour as the actions of the different classes would be controller in the tamagotchi class. Apart from this we also have an inventory class which just stores all of the inventory of food, toys, and types of medicine. These things in the inventory appropriately affect the stats of the tamagotchi which is created by the user.

- **The key solutions in our implementation -** There were certain things that we needed to implement in the game which were rather vital in regards to the functioning of the game in general. One main thing for implementation was the tamagotchi and the main class, where we used a switch statement for all the different options which are presented to the user. Apart from the main function we also have the different actions the tamagotchi can perform, and can interact with the different classes, so we have a feed(), play(), and medicate() function, these functions are the main access points for the pet actually performing an action and these actions have effects on the statistics of the current instance of the tamagotchi.

- **Path to main file -** src/main/java/softwaredesign/Main.java
- Path to JAR file -out/artifacts/software_design_vu_2020_jar/
- Link to youtube video demo - https://youtu.be/8-yaWi0jXUM

Time log:

	A	B	C	D
1	Tanav	Worked on inventory class and design pattern of class diagram	Week no - 8	4 hours
2	Filip	worked on main class and state machine		4 hours
3	Denay	worked on sequence diagram and implementation into code		4 hours
4	Sanam	worked on basic structure of code from class diagram and object diagram		4 hours
5				
6	Team number	43		
7				